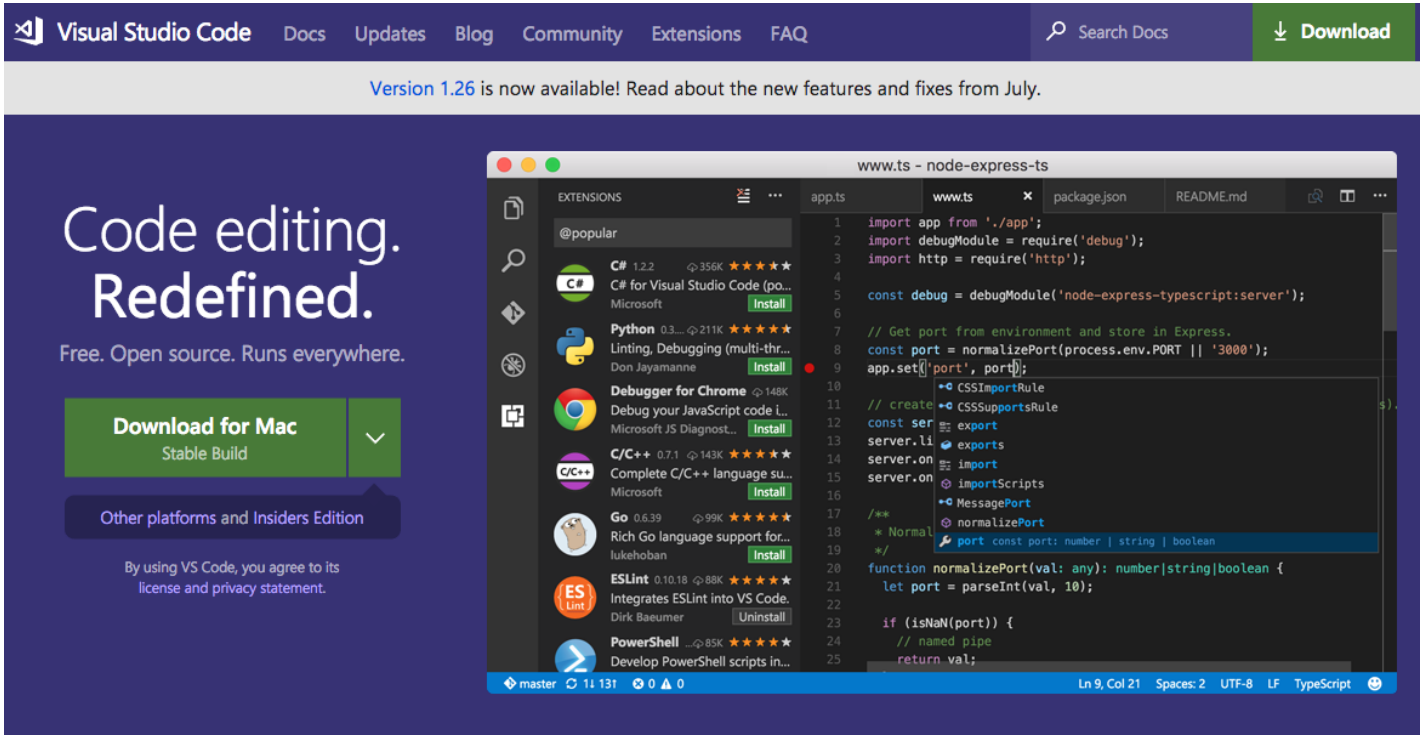
JavaScript

Index

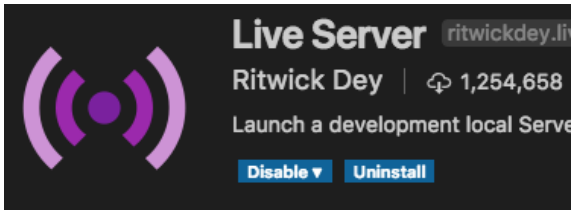
- Training Setup
- HTML
- Emmet
- CSS
- Javascript Part I: Language Elements
- Javascript Part II: Functional Programming
- Javascript Part III: Advanced programming

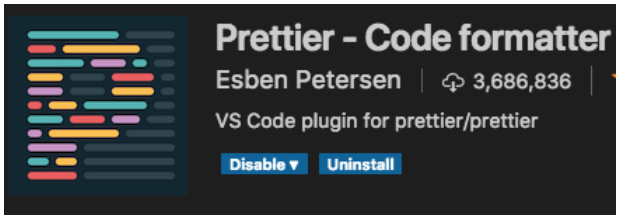
Environment Setup

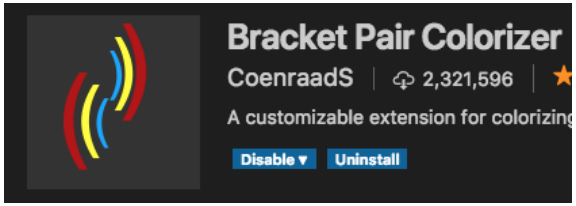
Setup – Visual Studio Code

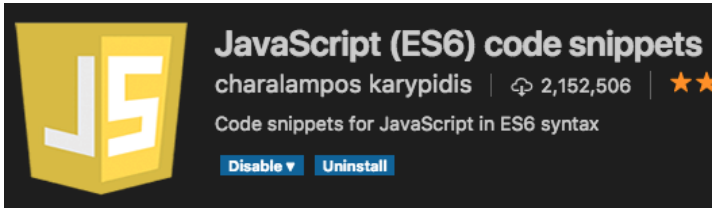


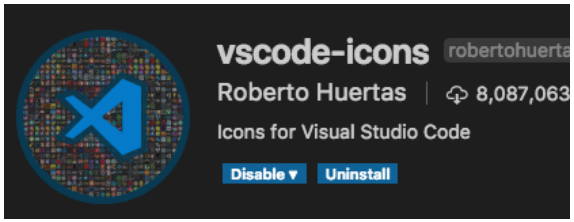
Setup – Visual Studio Code extensions

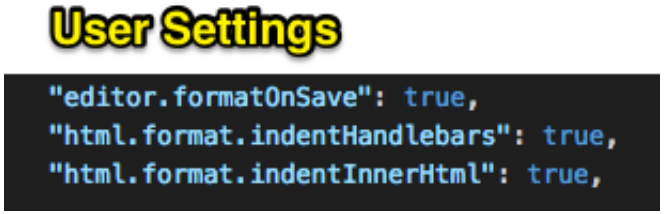








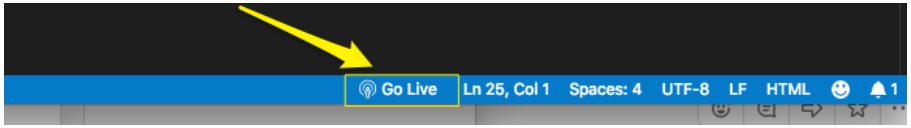




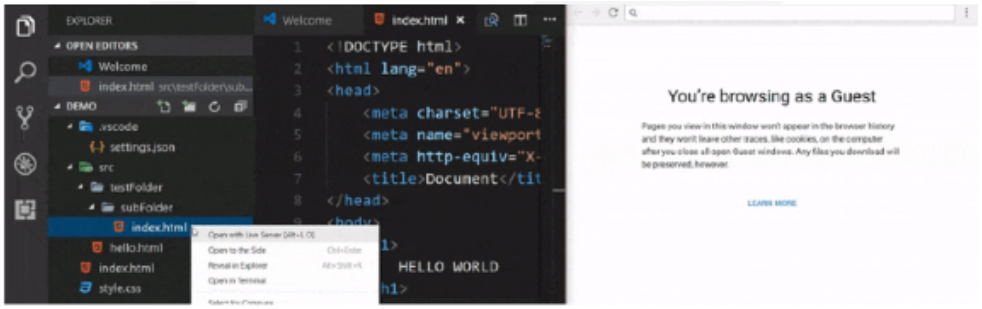
Live Server Extension

- Launch a development local Server with live reload feature

1. Open a project and directly click to Go Live from StatusBar to turn on/off the server



2. Rigth click on a HTML file and click “Open with Live Server”



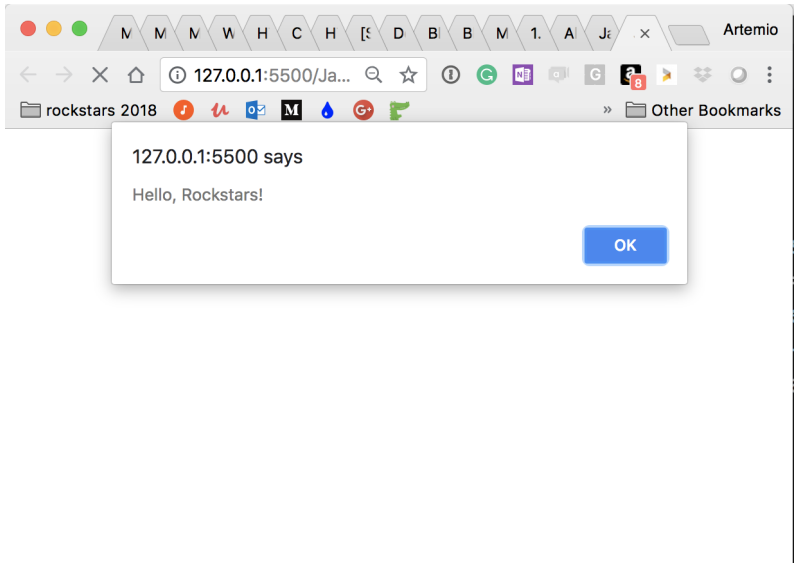
Chapter 1: Language Fundamentals

JavaScript

But, what is JavaScript after all?

Let the games begin: jump in!

```html
<!DOCTYPE html>
<html lang="en">

    <head>
        <meta charset="UTF-8">
        <meta name="viewport"
              content="width=device-width, initial-scale=1.0">
        <meta http-equiv="X-UA-Compatible" content="ie=edge">
        <title>JavaScript First Dive!</title>
    </head>

    <body>
        <script>
            alert('Hello, Rockstars!')
        </script>

    </body>

</html>
```

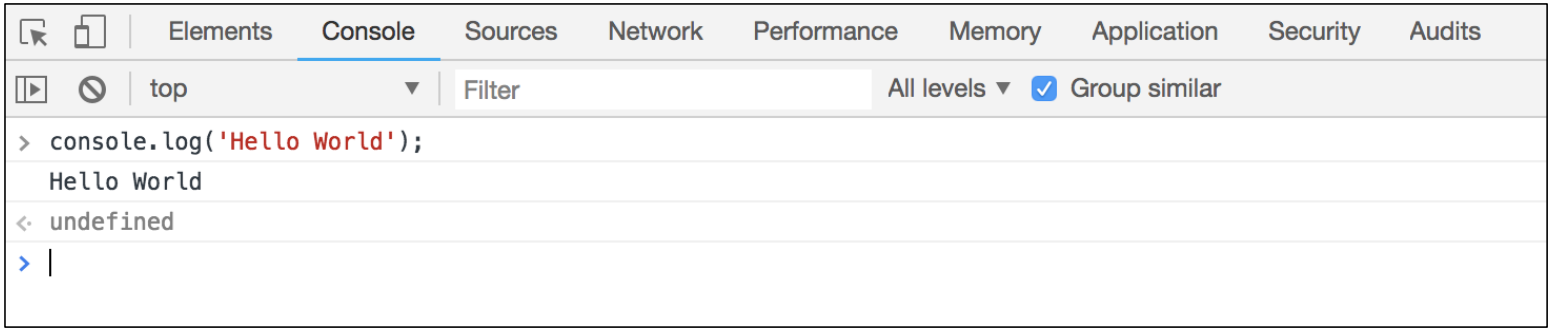


Built in web development tools: the console

- The five most popular web browsers (safari, IE, Edge, Chrome, Firefox, Opera) have built in tools.
- The elements allows to navigate the HTML DOM
- Web development tools commonly include a panel to debug scripts by allowing developers to add watch expressions, breakpoints, view the call stack, and pause, step over, step into, and step out of functions while debugging JavaScript
- The console is a common included tool, allowing to type in JavaScript commands and call functions, or view errors.
- The console is frequently used as output for debugging. This is the web development tool that will be used most during this training.

How to start the console

- **Google Chrome:** Ctrl + Shift + J (Win/Linx) or Cmd + Opt + J (Mac)
- **MS Edge or IE:** F12 to Developer Tools. Then navigate to Console Tab
- **Firefox:** Ctrl + Shift + K (Win/Linux) or Ctrl + Opt + K (Mac)



Variables in JavaScript

- Variables can ONLY start with letter, underscore or a dollar sign
- JavaScript is case sensitive
- JavaScript is a *loosely typed* language (or *dynamic*).
  - **Strongly typed languages**: must declare datatypes (c/c++, java, c#)
  - **Loosely typed languages**: not required to declare datatatypes
- The datatype of a variable is assigned when the variable receives a value (dinamically!)
- There are supersets of JS and addons to allow static typing (TypeScript, Flow)

Variables declaration

- Variables can be declared by any of these keywords *var, let* and *const*:
  - ***var*** *variableName (do not use anymore, just for backwards compatibility!)*
  - ***let*** *variableName*
  - ***const*** *variableName*
- ***var*** defines a variable global or locally to an entire functions, creating issues with block-level scoping
- var is an old feature (ES5), only included mostly for backwards compatibility. DO NOT USE!
- Since ES6 (EMACS 6) **let** and **const** were introduced to fix **var**
- ***let*** allows to modify initial value
- ***const*** does not allow to change value for *primitives*.
- Always use ***const***, unless you need to modify initial value of the variable

Variables – primitive datatypes

- Primitive datatypes: immutable, simple values, non-objects:
- JavaScript provides six primitive datatypes
  - **String** – sequence of text. Enclosed in “ or ‘ marks
  - **Number** – A number. Do not need quote marks around them
  - **Boolean** – a True or False value. The words *true* and *false* are reserved words
  - **Null** – always one value: ***null.***
  - **Undefined** – a variable without value has an ***undefined*** value
  - **Symbol** (introduced in ECMAS 6)
- The operator ***typeof*** retrieves the variable type.
- Further info: https://www.vojtechruzicka.com/javascript-primitives

Variables – no primitive datatypes

- Non Primitive DataTypes: objects with values *and* properties. They are mutable (can change)
  - Arrays – Structure that allows to store multiple values
  - Object Literals
  - Functions
  - Dates
  - Sets (ECMAS6)
  - Maps (ECMAS6)
  - Anything Else
- A variable can be used *before* being declared (?)

Variables - conventions

- Use **camelCase** convention for variable names
- Use character $ at the beginning of a variable name only to assign Jquery objects
- Use character _ at the beginning of a variable name only for patterns and advance models
- Use upper case character at the beginning of a variable name only for Objects (classes, constructors)
-  **const** does not define a constant value, but a constant reference to a value. Hence, a primitive value cannot change, but properties of a constant object can change

Variables – Object Wrappers

- What are the expected results for:
  - typeof new number(4);
  - typeof 4;
  - typeof new string('String');
  - typeof 'String';
  - typeof new Boolean(true);
  - typeof true
- What is the result of *null typeof* and why?
  - typeof null

Variables – Datatype conversion

- Convert *variable* To String:
  - String(*variable*)
  - *variable*.toString()

- Convert *variable* To Number:
  - Number(*variable)*
  - parseInt(*variable)*
  - parseFloat(*variable)*

- Not-a-Number or NaN, is a returned as error. It is an object.

Variables in JavaScript

- Create a JavaScript Sandbox
  - index.html
  - app.js
  - link app.js to index.html

Operators and The Math Object

- Artihmetic operators:
  - Addition(+)
  - Subtraction(-)
  - Division(/)
  - Multiplication(*)
  - Remainder(%)
  - Exponentiation(**)
  - Increment(++)
  - Decrement(--)
  - Unary negation(-)
  - Unary plus(+)

- The Math object *Math provides properties and methods for mathematical constants and functions:*
  - *Math.pi*
  - *Math.random()*
  - *Math.abs()*
  - *Math.round()*
  - *Math.ceil()*
  - *Math.floor()*
  - *Math.sin()*
  - *Math.sqrt()*
  - *Math.pow()*

String Methods and Concatenation

- Concatenation
  string1 + string2 + string3
- Append
  string1 += string2
- Method concat
  - *string.concat(string1,string2)*
- Escaping
  For keywords, use scape character **
  'This is a \'scape'\ example'
- Length
  - *string.length*

- Change Case (wrapper)
  string.toUpperCase();
  string.toLowerCase();
- Strings as arrays
  variable = 'I rock!'
  variable[0]
  variable[3]
- Search character
  string.indexOf('char');
  string.lastIndexOf('char');
  string.charAt('index')

Template Literals

- Are string literals allowing embedded expressions
- You can use multi-line strings and string interpolation with them
- Are enclosed by the back-tick (` `) character instead of the double or single quotes.
- Templates literals can contain placeholders indicated by dollar sign and curly braces *${expression}*
- In one phrase, it allows you to use **syntactic sugar** making substitutions more readable:

```
'Fifteen is '+(a+b)+' and\nnot '+(2*a+b)+'.';
`Fifteen is ${a+b} and not ${2*a+b} `
```

Arrays

- Arrays are a collection of objects. Are used to store multiple values in a single variable

```
const pets = ['dog','cat','ginea pig',parrot','hamster',fish'];
const countries = new Array('Germany','Russia','France',Mexico');
```

- Accessibles trhough an index number

```
const value = pets[0] ; // returns 'dog'
```

Length Properties

```
pets.length
```

Is Array?

```
Array.isArray(pets);
```

Adding element at the end

```
countries.push('USA');
```

Adding element at the begining

```
countries.unshift('Canada');
```

Splice Values

```
var = pets.length
```

Reverse

```
Array.isArray(pets);
```

Concatenate

```
countries.concat('USA');
```

Sort

```
countries.sort();
```

Object Literals

- A Java Script Object Literal is a comma-separated list of name-value pairs wrapped in curly braces.
- Objects literals encapsulate data, enclosing it in a tidy package.

```
const pet = {
    petName : 'Apollo',
    type : 'Dog',
    years : 2
};
```

- Can accept ***any*** data type, including array literals, functions, objects

Date and Time

- JavaScript Date objects are based on a time value that is the number of milliseconds since January 1st, 1970 UTC
- Sintax

```
new Date();
new Date(value);
New Date(dateString)1
New Date(year, monthIndex[,day[,hours[,minutes[,seconds[,milliseconds]]]]]);
```

- Date variable

```
const today = new Date('September 10, 2018 10:00:00');
```

- The argument *monthIndex* is 0-based, *January = 0* and *December = 12*

  today.getMonth( )
  today.getDay( )
  today.getFullYear

- Timestamp (amount of seconds that has passed since January 1st 1970)

  today.getTime()

Date and Time

- JavaScript Date objects are based on a time value that is the number of milliseconds since January 1st, 1970 UTC

- Set Specific time

```
birthday.setMonth(2);
birthday.setDate(12);
Birthday.setFullYear(1998)
```

- Further reference https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

Operators and Conditionals

- Logical operators are used to test for *true* or *false*

Comparison operators

==	*equal to*
===	*equal value and equal type*
!=	*not equal*
!==	*not equal value nor type*
>	*greater than*
>	*less than*
>=	*greater than or equal to*
<=	*less than or equal to*

Logical operators

&&	*and*
||	*or*
!	*not*

If Conditional: Sintax

```
if (conditional is true) {
    do something
} else {
    do something else
}
```

Ternary conditional: sintax

```
variable = (condition) ? valueWhenTrue : valueWhenFalse
```

Switches

```
switch(expression) {
    case condition1: statement(s)
    break;

    case condition2: statement(s)
    break;

    default: statement(s)
}
```

JavaScript Functions

- JavaScript is a *functional language,* hence understanding (*really understanding*) functions is critical
- Functional programming is a style of programming that's focused on solving problems by composing functions instead of specifying sequences of steps, as in *imperative programming.*
- Having functions as first-class objects is the first step toward *functional programming,* as they can be passed to functions as arguments.
- Functions can be passed as arguments to another function that might, a later point in application execution, call the passed-in function.
- This is an example of a concept known as *callback function*

JavaScript Functions

- In JavaScript, functions are *first-class objects* or *first-class citizens* because in JavaScript they possess all the capabilities of objects and are treated like any other object in the language.
- Functions *are* objects, just with an additional capability of being *invokables:* Functions can be called or invoked in order to perform an action

Assigned to variables, arrays and properties

```
var rockstarFunction = function() { };
rockstarArray.push(function() {});
rockstar.data = function() {};
```

Passed as argument to other functions

```
function call(rockstarFunction) {
   rockstarFunction();
}
Call(function(){});
```

Assigned to variables, arrays and properties

```
var rockstarFunction = function() { };
rockstarArray.push(function() {});
rockstar.data = function() {};
```

Passed as argument to other functions

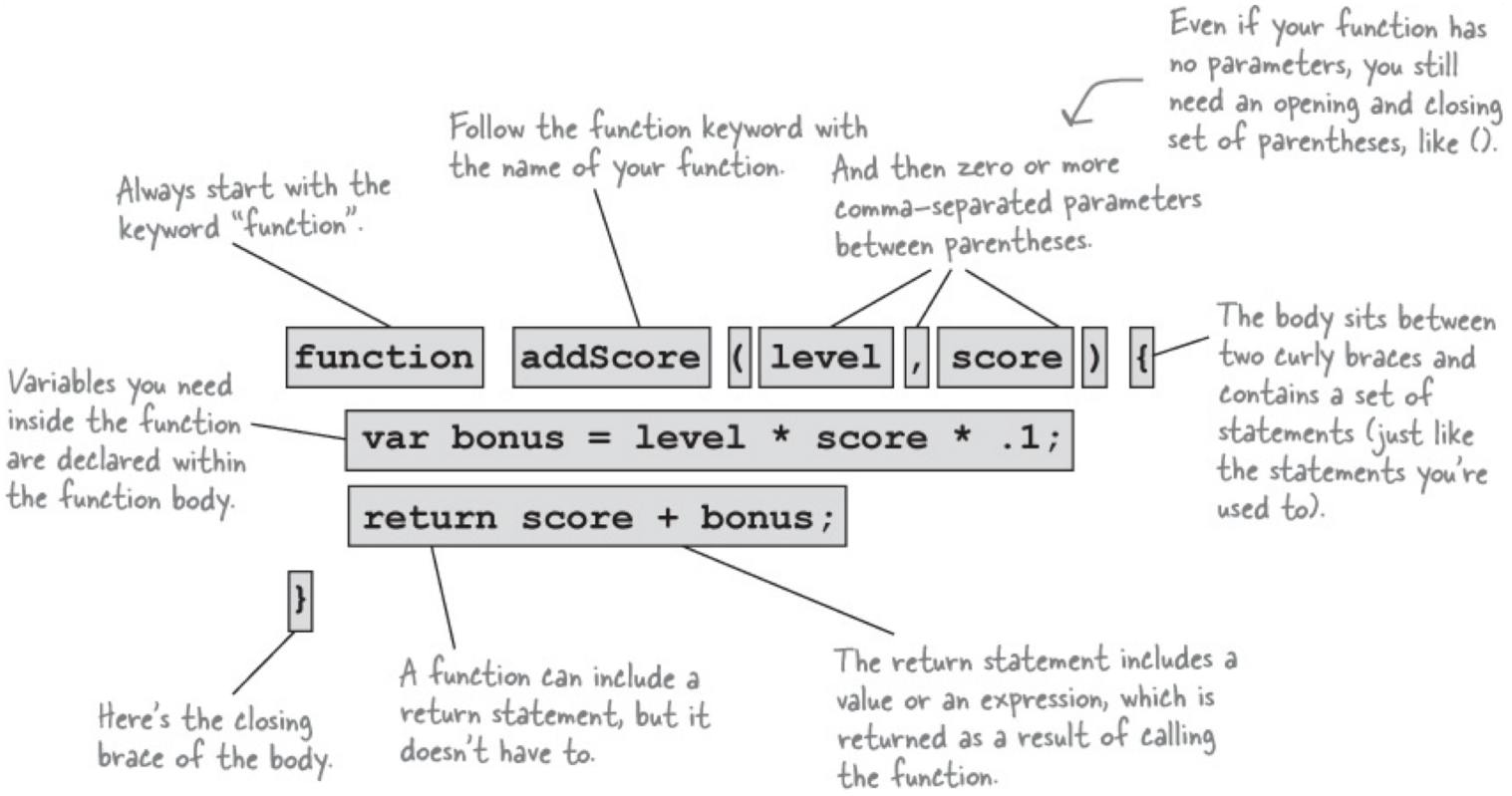
```
function call(rockstarFunction) {
   rockstarFunction();
}
Call(function(){});
```

Java Script – Functions!

JavaScript – The *other* Functions!

- Functions are the **key to the JavaScript programming style**.
- Like in many other programming languages, functions in JavaScript allow you to take a bit of code, give it a name and then refer to it over and over whenever it is needed.
- To use a function, it has to be defined first.
- There are two main *styles* for function definition:
  - Function Declarations
  - Function Expressions
- Also, Immediately-invoked function expression (IIFE pronounced *iffy*)

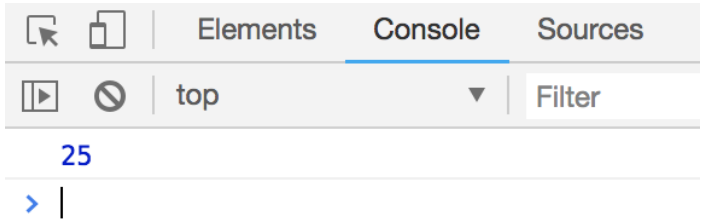
JavaScript – Anatomy



Function Declarations

- A **function definition**, (also called a **function declaration**, or **function statement**) consists of the *function* keyword followed by:
  - The name of the function
  - A list of parameters to the function, enclosed in parentheses and separated by commas
  - The JavaScript statement that defines the function, enclosed in curly brackets {}

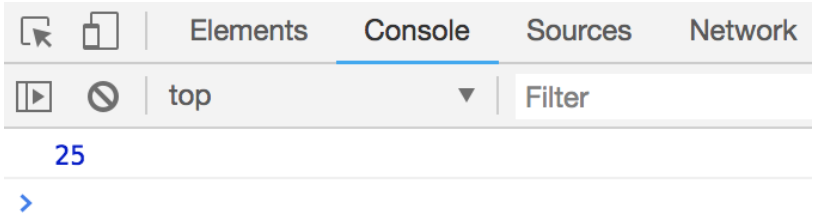
```
// Function Declarations
function square(number) {
  return number * number;
}
```



Function Expressions

- Functions can also be created by a *function expression.* Such function can be **anonymous;**
  - *const square = function(number) { return number * number;};*
- However, a name can be provided and can be used inside the function itself
  - *const factorial = function fac(n) {return n<2?1:n*fac(n-1);};*

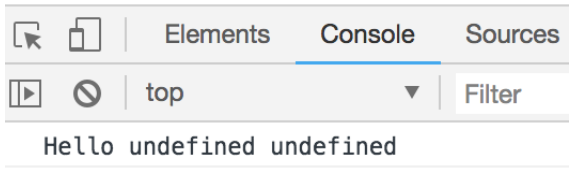
```
11  // Function Expression
12  const square = function(number) {
13    return number * number;
14  };
15
16  console.log(square(5));
```



Functions – Parameters

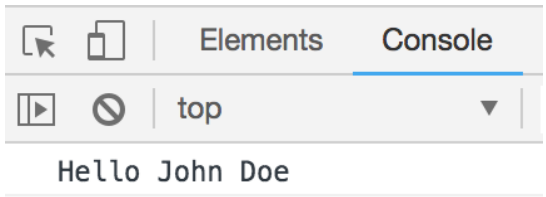
- Functions can accept parameters. Function expressions are convenient when passing a function as an argument to another function.
- What happens when a parameter is missing?

```
function greet(firstName, lastName) {
  return "Hello " + firstName + " " + lastName;
}
console.log(greet());
```



- ES6: default values parameters are defined a follows

```
function greet(firstName = "John", lastName = "Doe") {
  return "Hello " + firstName + " " + lastName;
}
console.log(greet());
```



Immediately Invoked Function Expressions

- IIFEs are functions that are executed immediately after defined
- They run only once and never more

```
(function ( ) {
    // logic here
}) ( )
```

- Why? Data privacy due to scope: any variable declared within the IIFE cannot be accessed by the outside world
- Very useful for Module Design Pattern (for ES5)

Functions within an object literal - Methods

- In Object Oriented Programming, a function inside and object is called *method*
- You can add a *function* to an Object Literal to create a *method*
- A methos can access a *property* by using keyword *this*

```
// PROPERTY METHODS for OBJECT LITERALS
const car = {
  brand: "nissan",
  year: 2011,
  started: false,
  clean: false,
  plate: "XTR1341",

  startEngine: function() {
    this.started = true;
  },
  stopEngine: function() {
    this.started = false;
  },
  cleanCar: function() {
    this.clean = true;
  },
  changePlate(newPlate = this.plate) {
    if (typeof newPlate) {
      this.plate = newPlate;
    }
  }
};
```

Loops and Iteration

- Most popular loops: *for* | *while* | *do-while*

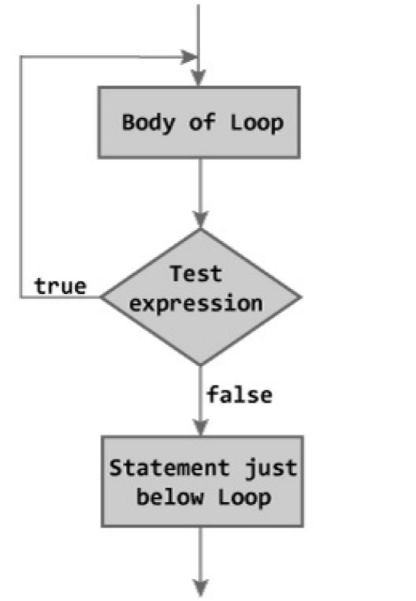


Figure: Flowchart of do...while Loop

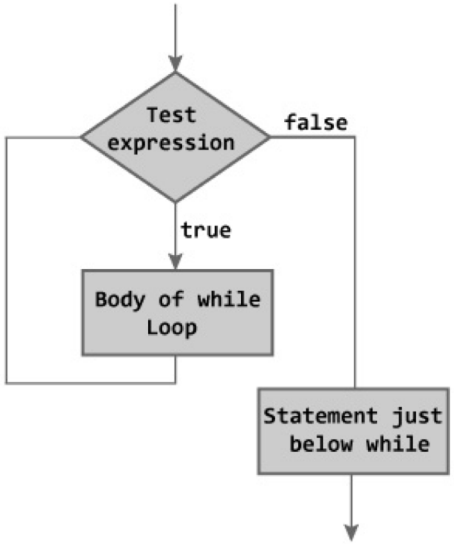


Figure: Flowchart of while Loop

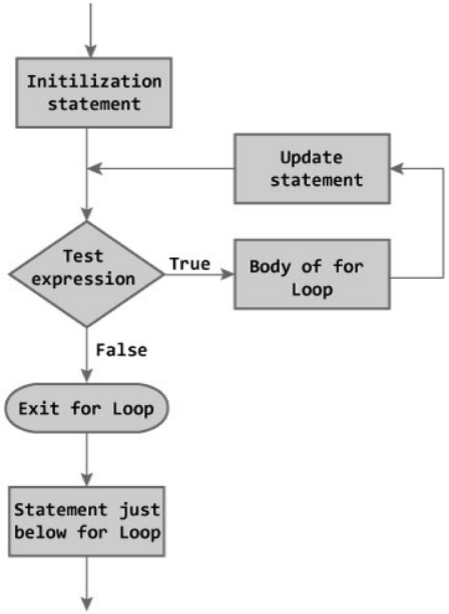
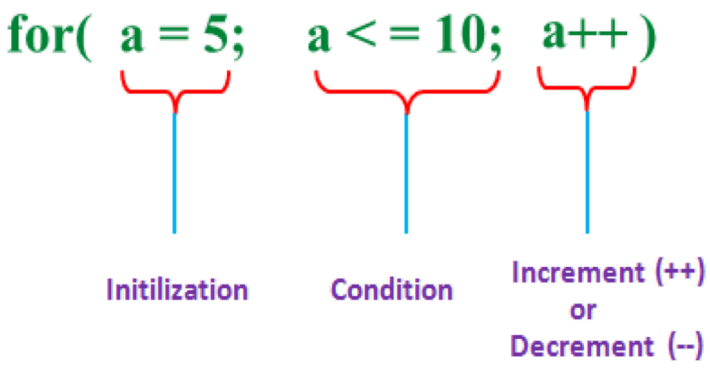
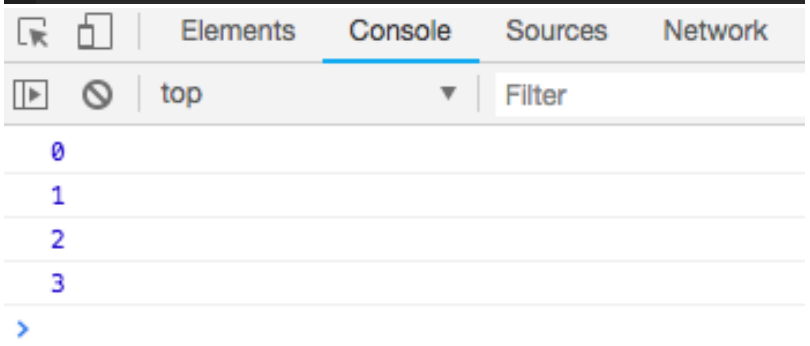


Figure: Flowchart of for Loop

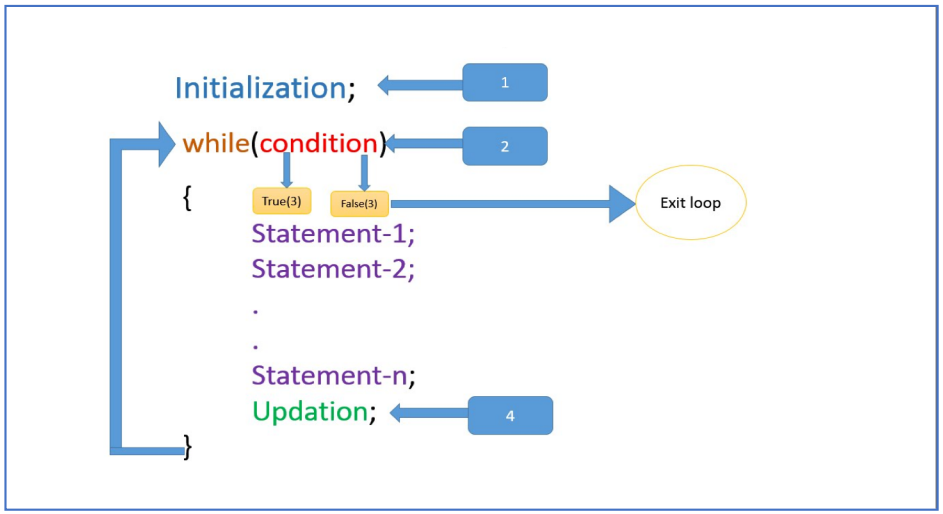
For loop



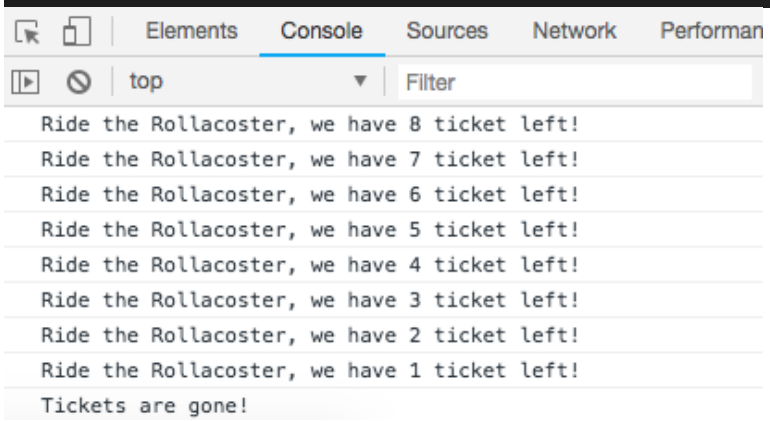

```
for (let i = 0; i <= 3; i++) {
  console.log(i);
}
```



While Loops




```
// WHILE LOOP
let tickets = 10;

while (tickets > 0) {
  console.log(`Ride the Rollacoster, we have ${tickets}
  tickets--;
}
console.log("Tickets are gone!");
```



Do … While Loops

- Repeat the statement *at least 1* then go for *do-while* loop.
- Sintaxis

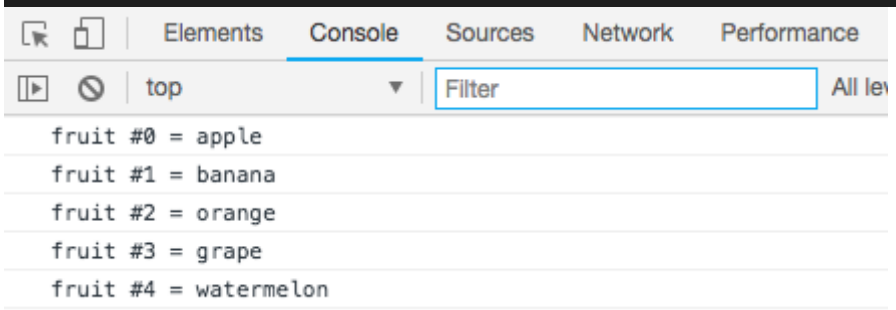
```
do {
// loop body
}
while (condition)
```

```
// DO .. WHILE
let scoop = 1;

do {
  console.log(`Add scoop #${scoop}`);
  scoop++;
} while (scoop <= 5);
{
  console.log(`This is a ${scoop - 1}-scoops icecream!`);
}
```

Array loops

- Special loop for arrays

```
// Arrays loops

const fruits = ["apple", "banana", "orange", "grape", "watermelon"];

fruits.forEach(function(fruit, index) {
  console.log(`fruit #${index} = ${fruit}`);
});
```



The map() Method

- It is not properly a loop, but kind of
- The map() method creates a ***new array*** with the results of calling a function for every array element
- The map() method calls the provided function code for each element in an array, in order
- Sintaxis:

```
array.map(function(currentValue, index, arr) this value);
```

thisValue: *optional. A value to be passed to the function to be used as it "this" value. If empty, then "undefined" will be passed as "this" value.*

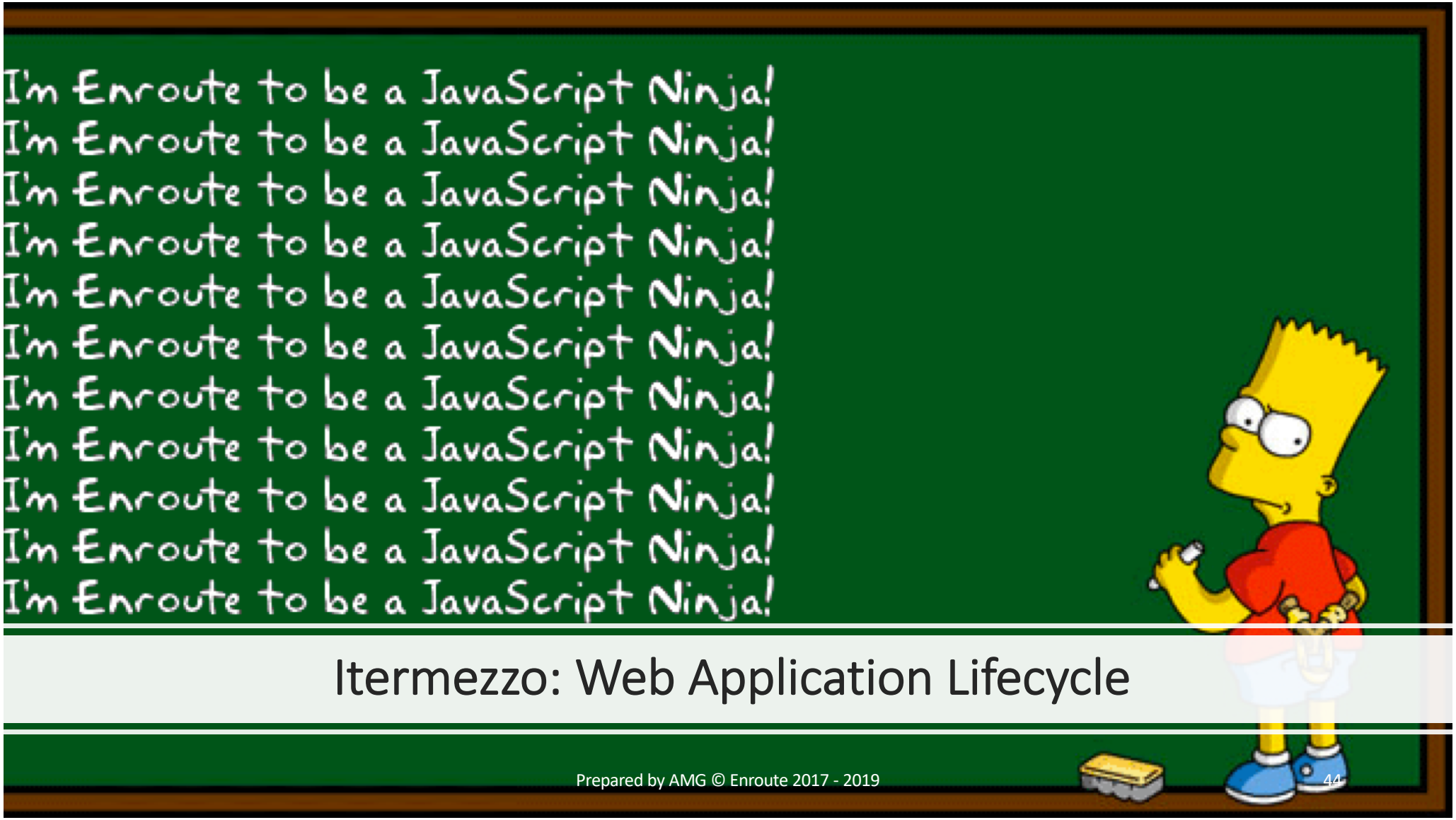
For … in Loop {for objects}

- The ***for … in*** statement iterates over all non-Symbol, enumerable properties of an object
- Syntax

```
for (variable in object) {
    // Do something
}
```

  - Variable: a different property name is assigned to *variable* on each iteration
  - Object: Objects whose non-symbol enumerable properties are iterated

- Not to be used to iterate over an Array where the index order is important



Itermezzo: Web Application Lifecycle

Web Application lifecycle

- This curse is focused in the context of client-side web application
- In this context, the browser is the engine that executes JavaScript code
- It is mandatory to have a clear understanding of the complete web application lifecycle and how JavaScript code fits into this lifecycle
- The lifecycle of a typical client-side web application begins with the user typing a URL into the browser's address bar, or by clicking a link
- The lifecycle of the application ends when the user closes the browser or leaves the webpage

Web Application Lifecycle

1. The user enters a URL into Web Browser

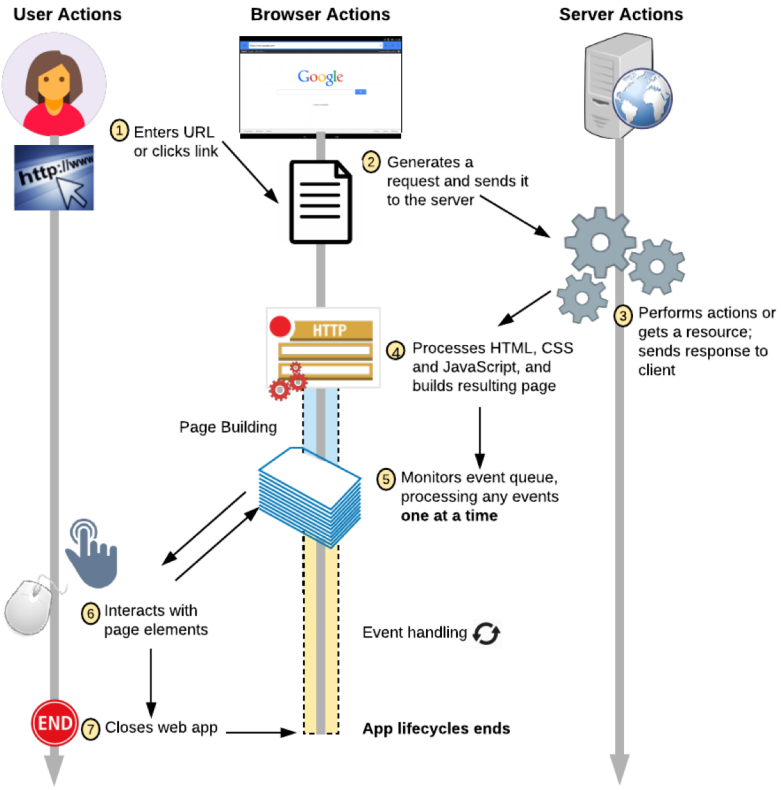
2. The browser send a request to a server

3. The server process the request and creates a response composed (usually) of HTML, CSS and JavaScript

4. The browser receives the response and setup the interface using the code from the server. This step is called **page building**

5 & 6 The browser enters in a loop (event loop) waiting for events to occur and starts invoking event handlers. These step is called **event handling**

7. The life cycle ends when the user closes or leaves the web page



Example: simple web app lifecycle

HTML

```html
<!--
    Shows the webapp lifecycle
    Enroute Rockstars (c) 2017 - 2018
-->
<!DOCTYPE html>

<html lang="en">

    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=devi
        <meta http-equiv="X-UA-Compatible" conter
        <link rel="stylesheet" href="style.css">
        <title>LifeCycle Example</title>
    </head>

    <body>
        <ul id="first"></ul>
        <ul id="second"></ul>

        <script src="app.js"></script>
    </body>

</html>
```

CSS

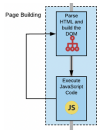
```css
#first {
  color: green;
  font-size: 150%;
}
#second {
  color: red;
  font-size: 150%;
}
```

JavaScript

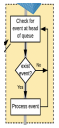
```javascript
function addMessage(element, message) {
  const messageElement = document.createElement("li");
  messageElement.textContent = message;
  element.appendChild(messageElement);
}

const first = document.getElementById("first");
const second = document.getElementById("second");

addMessage(first, "Page Loading");
// Catches "mousemove"
document.body.addEventListener("mousemove", function() {
  addMessage(second, "Event: mousemove");
});
// Catches "mousedown" event
document.body.addEventListener("mousedown", function(e) {
  if (e.button === 0) {
    msg = "Left Click";
  } else if (e.button === 1) {
    msg = "Wheel click";
  } else if (e.button === 2) {
    msg = "Right Click";
  } else {
    msg = "Unknown button!";
  }
  addMessage(second, `Event: click ${msg}`);
});
```

Web Applications are GUI Applications

- Client-side Web Applications are Graphical User Interface (GUI) Applications (desktop applications or mobile applications)
- GUIs applications' lifecycle phases are carried out in the following **two phases:**



PHASE I : Page Building – Set up the user interface

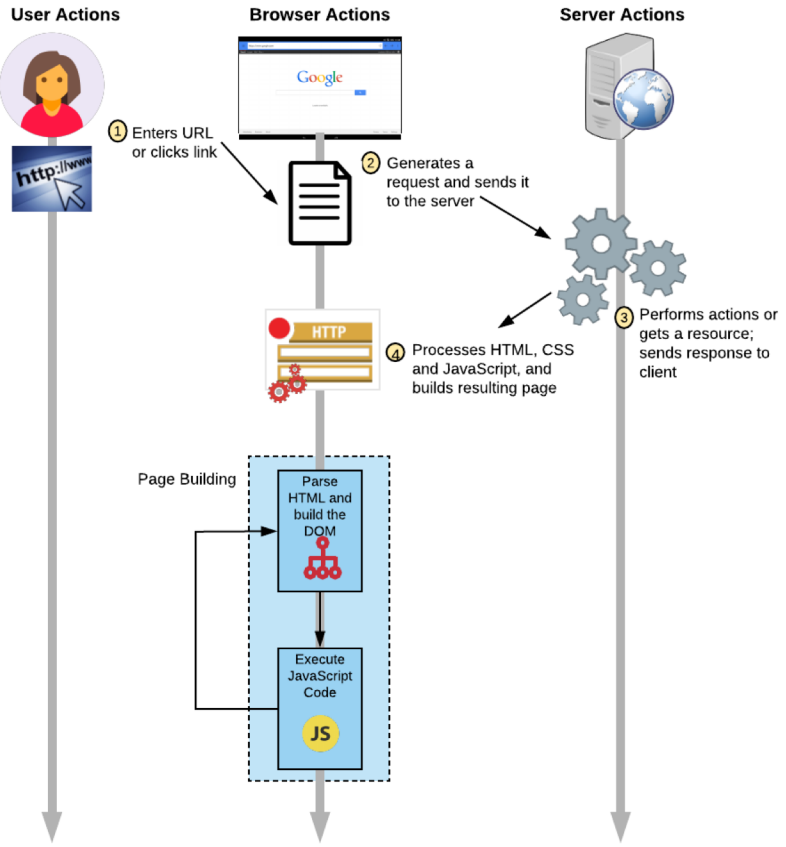


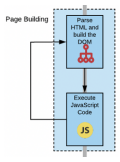
PHASE II : Event handling – Enter a loop, waiting for events to occur, and start invoking event handlers

- Let's review these two important phases in detail

The page-building phase

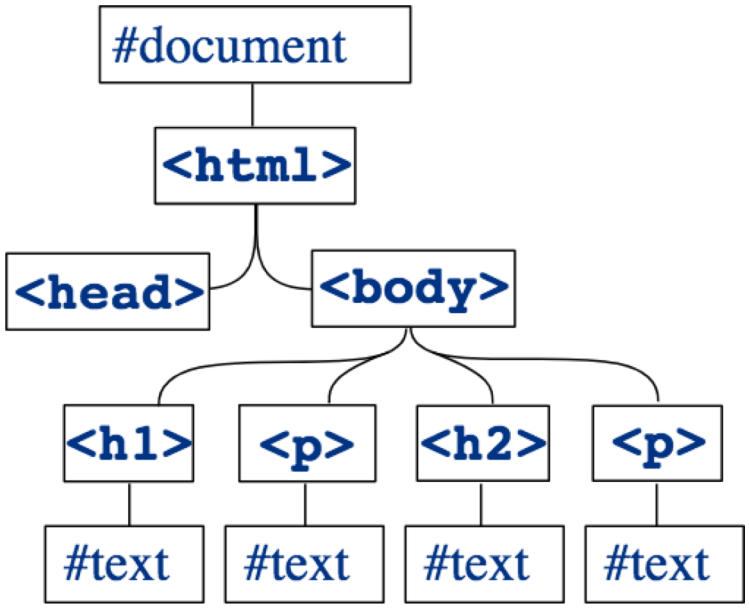
- Before anything, the page has to be built from HTML, CSS and JavaScript code received from the server. The goal is to set up the UI of a web app, this is done in two steps:
- **Step 1: Parsing the HTML and building the DOM.** It is performed when the browser process HTML nodes
- **Step 2: Executing JavaScript code.** It is performed whenever a special HTML element *<script></script>* is encountered.
- During this phase, the browser can switch between these two steps as needed

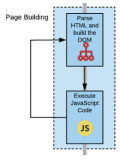




STEP 1: Parsing HTML and building the DOM

- The browser builds the UI by parsing one HTML element at a time, and building a **DOM** (**D**ocument **O**bject **M**odel)
- The DOM is an API for HTML and XML documents (a web page is a document)
- It provides a structured representation of the HTML page, and a programatic way to modify its content and visual representation
- The DOM treats an HTML document as tree structure in which every HTML element is represented as a node.





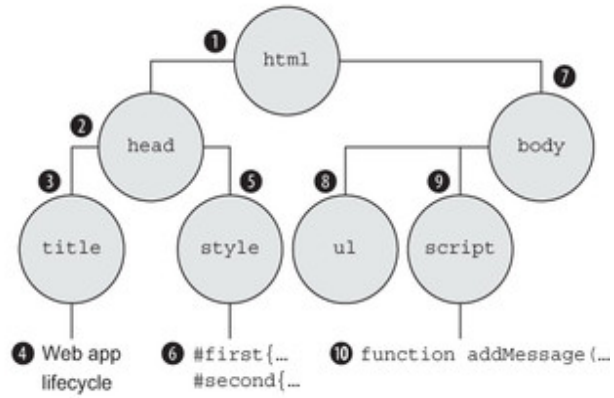
STEP 1: Parsing HTML and Building the DOM

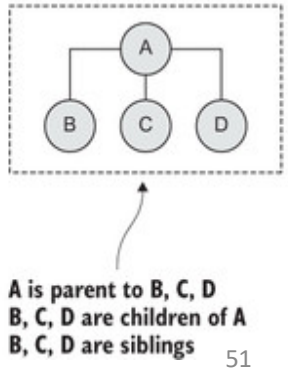
- This is the DOM of the example page *before* the JavaScript code is reached
- By the time the browser encounters the first *script* element, it has already created a DOM with multiple HTML elements (the nodes or the right)

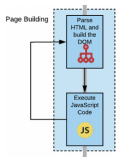
HTML code

```
<!DOCTYPE html>
1 <html>
2   <head>
3     <title> Web app lifecycle </title>
5     <style>
6       #first { color: green;}
        #second { color: red;}
      </style>
    </head>
7   <body>
8     <ul id="first"></ul>
9     <script>
      function addMessage(element, message){
        var messageElement = document.createElement("li");
10      messageElement.textContent = message;
        element.appendChild(messageElement);
      }
      …
```

DOM: built from the HTML code!

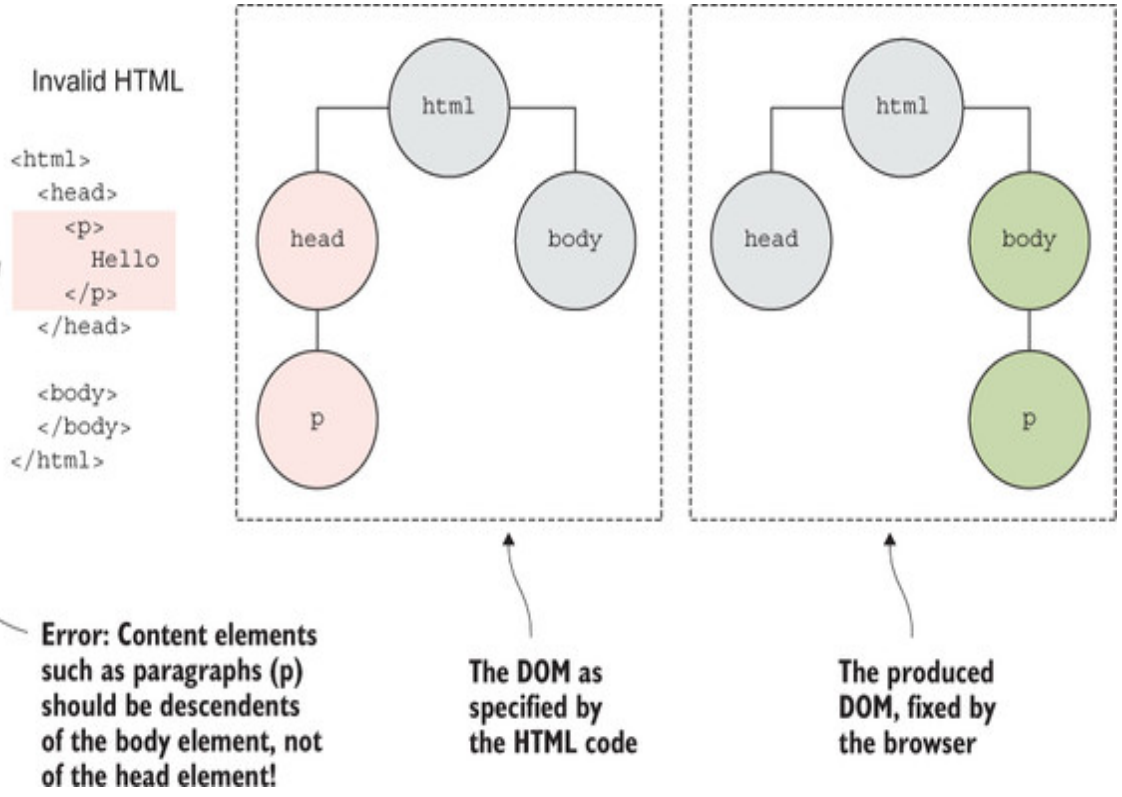


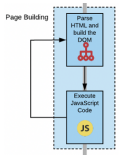




STEP 1: Parsing HTML and Building the DOM

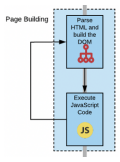
- Even though the HTML and the DOM are closely linked, they aren't one and the same.
- The browser can even fix problems that it finds with the HTML *blueprint* in order to create a valid DOM





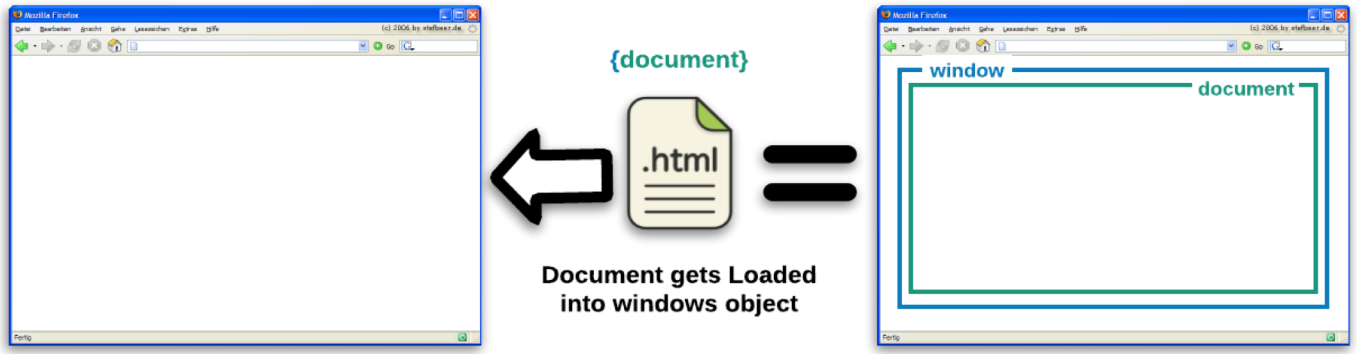
STEP2: Executing JavaScript code

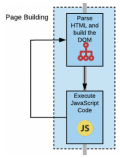
- During page construction, when the browser encounters the *script* element, it pauses the DOM construction from HTML code and starts executing JavaScript code
- JavaScript code is executed by the **browser's JavaScript engine**;
  - Firefox: SpiderMonkey (first JavaScript engine)
  - Google Chrome: V8,
  - Microsoft Edge/Internet Explorer: Chakra
  - Apple Safari: Nitro
- The browser provides an API through **a global object** that can be used by the JavaScript engine to interact with and modify the page.



STEP2: Executing JavaScript code - detour

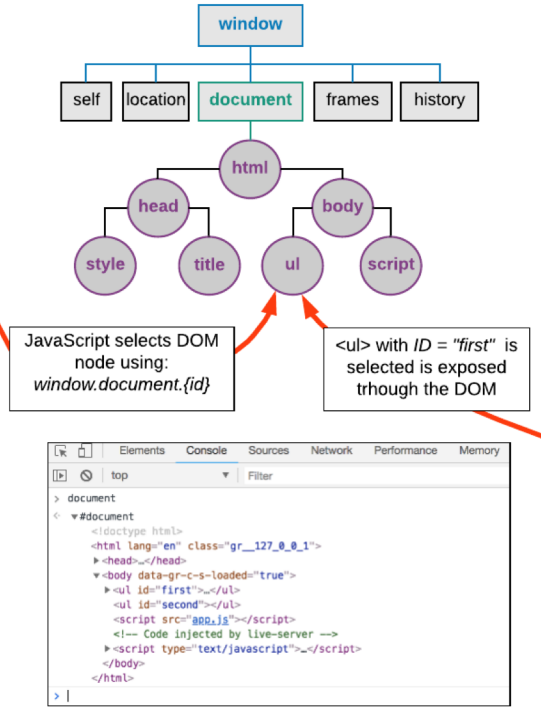
- The primary global object that the browser exposes to the Java Script Engine is the ***window*** object.
- The ***window*** object is **the** global object. Through it it is possible to access other global objects, global variables and browser APIs
- The property ***document*** represents the DOM of the current page





STEP2: Executing JavaScript code - detour

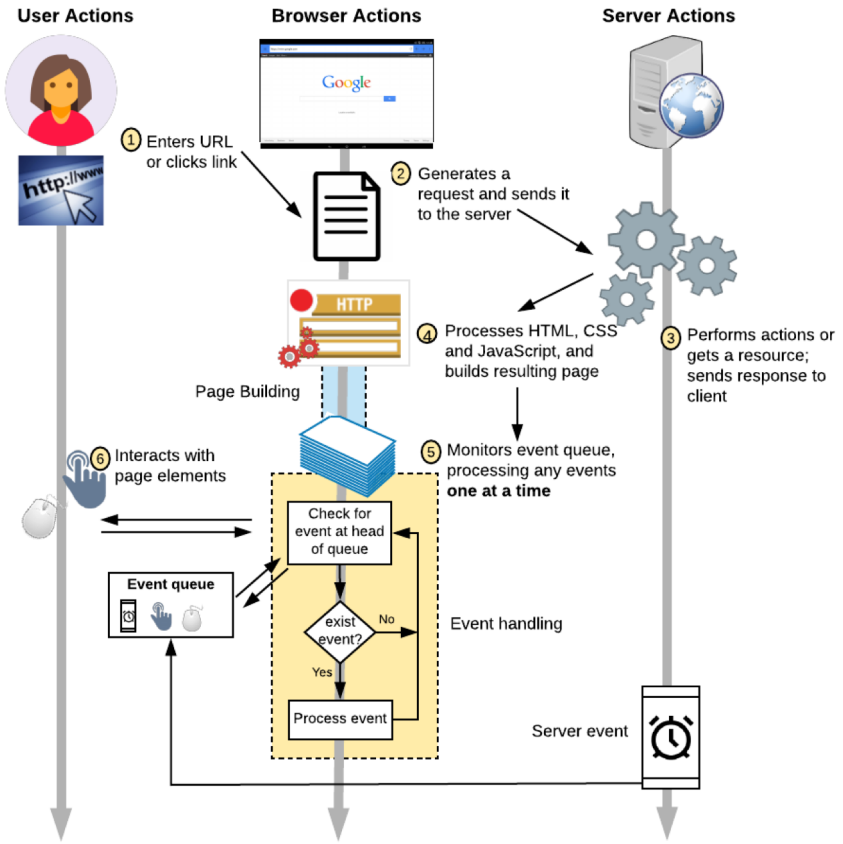
```
function addMessage(element, message) {
  const messageElement = document.createElement("li");
  messageElement.textContent = message;
  element.appendChild(messageElement);
}

const first = document.getElementById("first");
const second = document.getElementById("second");

addMessage(first, "Page Loading");
// Catches "mousemove"
document.body.addEventListener("mousemove", function() {
  addMessage(second, "Event: mousemove");
});
// Catches "mousedown" event
document.body.addEventListener("mousedown", function(e) {
  if (e.button === 0) {
    msg = "Left Click";
  } else if (e.button === 1) {
    msg = "Wheel click";
  } else if (e.button === 2) {
    msg = "Right Click";
  } else {
    msg = "Unknown button!";
  }
  addMessage(second, `Event: click ${msg}`);
});
```

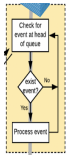



```
<!--
    Shows the webapp lifecycle
    Enroute Rockstars (c) 2017 - 2018
-->
<!DOCTYPE html>

<html lang="en">

    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=devi
        <meta http-equiv="X-UA-Compatible" conten
        <link rel="stylesheet" href="style.css">
        <title>LifeCycle Example</title>
    </head>

    <body>
        <ul id="first"></ul>
        <ul id="second"></ul>

        <script src="app.js"></script>
    </body>

</html>
```

The event-handling phase

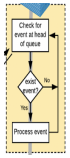
- The browser execution environment is a *single-threaded* execution model: **executes one piece of code at a time**
- During the event-handling phase, all events are queued up as they occur
- The events in the queue are processed one by one, by the *single-threaded* browser execution environment





The event-handling phase

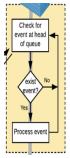
- The event handling process flows as shown in the previous slide
- Events are *asynchronous*: can occur at unpredictable time and order
- The vast majority of code executes as a result of an event
- Event handling is central to web applications
- Before an event can be handled, our the code has to notify the browser that we are interested in handling a particular event.
- This is called ***event-handler registration.***



The event-handling phase

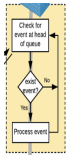
- Even though it is possible to assign functions to special object properties in order to register, the recommended way to do it is by using the method *addEventListener*
- The *addEventListener* method enables us to register as many event-handler functions as needed
- In this example, the code registers two handlers for two events: *mousemove* and *mousedown*

```
// Catches "mousemove"
document.body.addEventListener("mousemove", function() {
  addMessage(second, "Event: mousemove");
});
// Catches "mousedown" event
document.body.addEventListener("mousedown", function(e) {
  if (e.button === 0) {
    msg = "Left Click";
  } else if (e.button === 1) {
    msg = "Wheel click";
  } else if (e.button === 2) {
    msg = "Right Click";
  } else {
    msg = "Unknown button!";
  }
  addMessage(second, `Event: click ${msg}`);
});
```



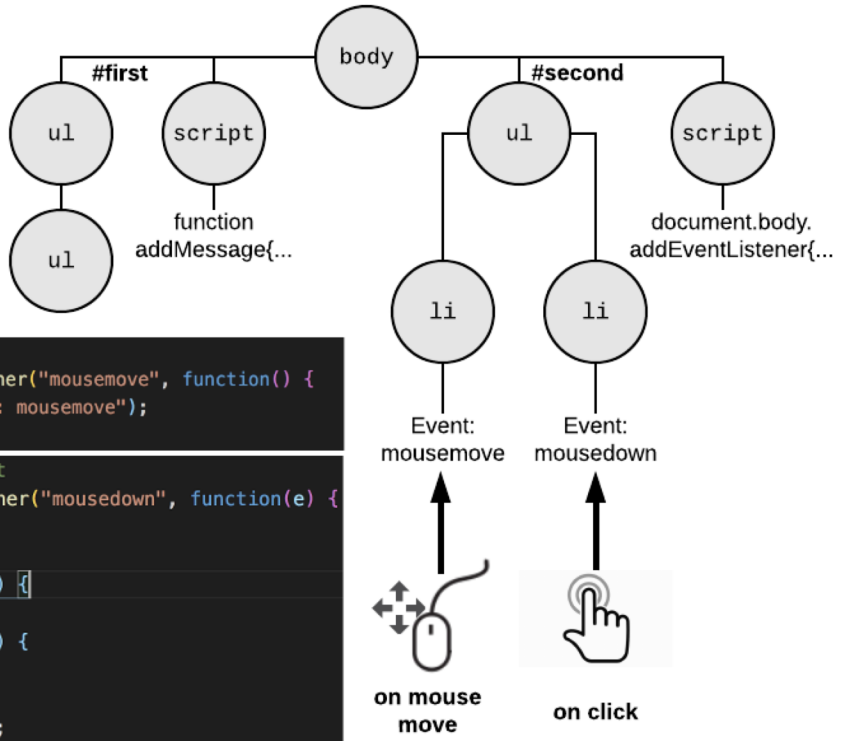
The event-handling phase

- The main idea behind event handling is that when an event occurs, the browser calls the associated event handler
- Only a single event handler can be executed at once. Any following events are processed only after the execution of the current event handler is fully complete
- In the example used, how the DOM is influenced by the *movedown* event?
- The execution of the handler function selects the second list (id="second") and, by using the *addMessage* function, adds a new list item (new DOM node)



The event-handling phase

- The DOM of the example application, after handling the events:
  - *mousemove*
  - *mousedown*
- Both, *#first* and *#second* elements are modified by JavaScript code




```
// Catches "mousemove"
document.body.addEventListener("mousemove", function() {
  addMessage(second, "Event: mousemove");
});
```

```
// Catches "mousedown" event
document.body.addEventListener("mousedown", function(e) {
  if (e.button === 0) {
    msg = "Left Click";
  } else if (e.button === 1) {
    msg = "Wheel click";
  } else if (e.button === 2) {
    msg = "Right Click";
  } else {
    msg = "Unknown button!";
  }
  addMessage(second, `Event: click ${msg}`);
});
```

Quiz

- Does the browser always build the page exactly according to the given HTML?
- Why must browsers use an event queue to process events?
- What are the two phases in the lifecycle of a client-side web application?
- What is the main advantage of using the *addEventListener* method to register an event handler versus assigning a handler to a specific element property?
- How many events can be processed at once?
- In what order are events from the event queue processed?